Consider open addressing with linear probing and an attempt to see whether a value e is in the set. If e hashes to h, then buckets with indexes h % b.length, (h+1) % b.length, (h+2) % b.length, ... are probed until either e is found or a bucket containing null is found.

e1 e2 e3 e4

0 1 2 3 4 5 6 7

b

Linear probing can result in *clustering*: many values occupy successive buckets, as shown to the right, leading to excessive probes to determine whether a value is in the set. Here, e1 could have hashed to bucket 2, then e2 and e3 to bucket 3, and finally e4 to bucket 2. If e5 hashes to 2,  
then five probes are necessary to determine that e5 is not in the set.

Several ways of reducing collisions have been proposed over the years. We outline some of them below just to give you a greater sense of the lengths people go to in attempting to make improvements in data structures. Search the internet and you are likely to find even more attempts to reduce collisions.

In preparation for the introduction of different ways of solving collisions, we assume that the value being hashed hashes to h. Linear probing probes the following buckets until null or the desired value is found —*remember*, *all integers below are taken mod the table size*, although we don't show that explicitly:

h, h+1, h+2, h+3, h+4, h+5, ...

**Quadratic probing**

Quadratic probingadds successive values of a polynomial to h: The simplest example uses this sequence:

h, h+1^2, h+2^2, h+3^2, h+4^2, ..., i.e. h+1, h+4, h+9, h+16, ...

Use this sequence of probes instead of linear probing on the example shown above (e1 hashes to 2, e2 and e3 hash to 3, and e4 hashes to 2), and values would be placed as shown to the right. Value e4 is now separated from e1, e2, and e3.

e1 e2 e3 e4

0 1 2 3 4 5 6 7

b

In general, one can choose any polynomial. For example, one could use this sequence of probes: where the buckets to be probed are numbered p(0), p(1), p(2), p(3), ..., and p(k) = h + 2k + 5k^2:

h, h+2+5, h+4+5\*4, h+6+5\*9, ...

Quadratic probing can reduce the number of collisions. But a big problem is ensuring that the probe sequence will cover all buckets to always find null if the value being probed for is not in the hash table. For example, suppose array b has size 8, as shown in our examples, and suppose a value *e* hashes to 0. Then, the quadratic probe sequence first shown above taken modulo the table size is as follows; some buckets are repeated many times.

0, 1, 4, 1, 0, 1, ...

Many papers have been written on quadratic probing, describing not only what quadratic polynomial to use but also the table sizes to use with that polynomial.

**Double hashing**

Use a second hash function hash(e) to help determine the probe sequence. Suppose e initially hashes to h and H = hash(e) Then use the probe sequence:

h, h +H, h + 2H, h + 3H, h + 4H, h + 5H, ...

**Cuckoo hashing**

The simple variant of cuckoo hashing uses two hash functions h1 and h2. To determine whether a value e is in the table, check the two positions b[h1(e)] and b[h2(e)] (taken modulo the table size, of course). If neither contains e, then e is not in the table; there is no need to worry about collisions. This is worst-case time O(1).

Similarly, to remove e from the table, look at those two buckets. If neither is e, then e is not in the set and nothing need be removed. If one of them is e, then set that bucket to null. This is worst-case time O(1).

Both search and remove take constant time in the worst case! How can that be! Well, all the work to allow those to be constant time takes place in the method to insert e into the hash table. It is shown to the right. *All indexes are taken module the table size*; to save space, we leave that implicit.

1. void insert(e) {

2. if (b[h1(e)] == e || b[h2(e)] == e) return;

3. int pos= h1(e);

4. // inv: Trying to insert e at b[pos]

5. loop n times: {

6. if b[pos] == null { b[pos]= e; return; }

7. Swap e and b[pos];

8. if (pos == h1(e)) pos= h2(e);  
9. else pos= h1(e);

10. }

11. rehash; insert(e);

12.}

The major change from earlier probing strategies is that this strategy will push one value out of the table (and place it elsewhere) to make room for the new one. That is what gives the method its name. In some species of cuckoo, the cuckoo chick pushes other eggs out of the nest when it hatches.

We discuss method insert. Suppose line 3 is executed, so that pos is the index of the bucket into which e is to be placed. We now have a loop that iterates at most n times, because pushing one egg out of the nest (one value out of the table) will require putting that egg into the table elsewhere. We don't discuss what value n is; that is